This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

Speculative Barriers with Transactional Memory

Manuel Pedrero, Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata

Abstract—Transactional Memory (TM) is a synchronization model for parallel programming which provides optimistic concurrency control. Transactions can run in parallel and are only serialized in case of conflict. In this work we use hardware TM (HTM) to implement an optimistic *speculative barrier* (SB) to replace the lock-based solution. SBs leverage HTM support to elide barriers speculatively. When a thread reaches an SB, a new *SB transaction* is started, keeping the updates private to the thread, and letting the HTM system detect potential conflicts. Once the last thread reaches the corresponding SB, the speculative threads can commit their changes. The main contributions of this work are: an API for SBs implemented with HTM extensions; a procedure to check the speculation state in between barriers to enable SBs with non-transactional codes; a HTM SB-aware conflict resolution enhancement where SB transactions stall on a conflict with a standard transaction; and a set of SB use guidelines derived from our experience on using SBs in a variety of applications. We evaluated our proposals in two different architectures with a full-system simulator and an IBM Power8 server. Results show an overall performance improvement of SBs over traditional barriers.

Index Terms—Speculative Barriers, Hardware Transactional Memory, Shared-Memory Parallelism, IBM Power8, GEMS.

1 Introduction

Transactional Memory (TM) [1], [2], [3] has emerged as an alternative to locking techniques to simplify parallel programming. A transaction is a section of code that is guaranteed to be executed atomically and in isolation. The TM system executes transactions speculatively in parallel while keeping track of memory accesses to detect and resolve conflicts. Thus, TM is considered to provide optimistic concurrency control as opposed to the pessimistic lock-based way of dealing with critical sections, which always serializes the execution. Many TM proposals have arisen in the last two decades, including both software (STM) [4], [5] and hardware designs (HTM) [6], [7], [8]. In the last years, major manufacturers of commercial processors have added HTM extensions to their architectures [9], [10], [11]. Such extensions are called *best-effort* HTMs because they do not offer finalization guarantees for transactions.

In this work we use such commercial HTM extensions to implement an optimistic *speculative barrier* (SB) to replace the pessimistic lock-based traditional solution. SBs leverage HTM support to allow threads to elide barriers by executing speculatively. When a thread reaches an SB, a new transaction (SB transaction) is started, which keeps the updates private to the thread while the HTM system monitors memory accesses for potential conflicts. Once the last thread reaches the corresponding SB, the speculative threads can commit their changes. SBs implement a partial order among transactions before the barrier and SB transactions, where the latter cannot commit until the former have finished.

Barrier-intensive applications [12] may exhibit poor speedups due to (i) load imbalance, where faster threads have to wait to slower ones; and (ii) barrier communication latency, specially on multi-chip processors. SBs are devised for the faster threads to speculate across the barrier instead of idling. Thus, load imbalance can be harnessed to hide the

• *All authors are with the Department of Computer Architecture, University of Malaga, Spain, 29071.*
E-mail: {mpedrero,quislant,eladio,zapata,oplata}@uma.es

Manuscript received November 22, 2019.

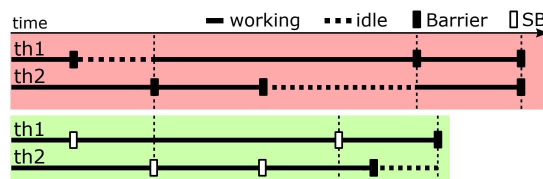

Fig. 1. Standard barriers (top) versus SBs (bottom).

barrier latency and getting ahead with work. Fig. 1 shows a scenario where such an imbalance causes the standard barriers to lag (top) while SBs (bottom) perform better even though the faster thread has to wait at the last barrier. Fig. 1 assumes no memory conflicts between threads.

One hard requirement for our implementation is HTM support for *escape actions* [13]. Escape actions allow non-transactional accesses executed inside transactions. We use them to enable communication among SB transactions without triggering aborts. SB control variables must be accessed inside transactions and a conflict might be detected if a transaction reads a variable that was updated by another transaction, unless such accesses are escaped.

This work extends a preliminary study on optimistic barriers [14] with the following contributions:

- We provide an *application programming interface* (API) for SBs implemented with commercial HTM extensions. The API includes begin/commit transaction wrappers as well as a barrier substitute.
- We add a procedure for *last barriers* which ends speculation of previous barriers and does not start new speculation.
- We introduce a *check speculation* mechanism to support the use of SBs with non-transactional codes. This mechanism checks the speculation state in between barriers and can also be used with transactional codes to increment the SB checkpoints.
- We propose a new HTM *SB-aware conflict resolution enhancement* where the SB transaction stalls on a conflict with a standard transaction, to be used instead

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

of the commonly used requester-wins/loses conflict resolution policy.
- We outline some SB *use guidelines* derived from our experience on using SBs in a variety of barrier-intensive applications.

We evaluated our proposals in two different architectures with support for HTM and escape actions: a simulator with GEMS [15] and an IBM Power8 server (Intel's HTM system does not provide escape actions). Results show an overall performance improvement of SBs over traditional barriers. The gains increase with the number of threads, specially in the 2-socket Power8 machine when there is inter-socket communication.

1.1 Background on Hardware Transactional Memory

Commercial HTM extensions [10], [11] implement a TM system based on caches and the coherence protocol, such as the seminal approach of Herlihy and Moss [1]. Core's private L1 cache is used to keep transactional updates, while L2 and L3 caches are used to keep track of transactionally read locations. Transactional read/write bits are provided with each cache block so that the coherence protocol can check for conflicts among transactions. On a transactional read request, the coherence protocol checks these transactional bits. If the write bit was set by another transaction, the protocol sends an abort command either to the requester (requester-loses conflict resolution policy) or to the requested thread (requester-wins conflict resolution policy). Requests from non-transactional code to transactional blocks favor non-transactional code enforcing the strong-isolation property [16].

On the Instruction Set Architecture (ISA) level, HTM extensions provide a concise set of instructions to manage transactions: *xBegin(ret)* marks the beginning of a transaction and may receive a label, *ret*, that points to the line of code we want the thread to resume on abort. From *xBegin()* onwards, the HTM system starts the bookkeeping of memory accesses in the cache hierarchy; *xCommit()* marks the end of a transaction, resets transactional bits and releases private updates. An abort instruction explicitly aborts a transaction from within. HTMs with support for escape actions add the pair *escapeBegin()* and *escapeEnd()* to temporally disable TM tracking in the code enclosed between them.

This HTM implementation results in a *best-effort* system where a transaction aborts whenever it runs out of bookkeeping hardware, risking live-lock. To ensure forward progress the user must implement a fallback mechanism which often consists of the transaction body protected by a global lock. Fig. 2 shows a simplified API for transactions that implements a fallback mechanism. The begin/commit instructions are replaced with the TX_START/TX_STOP wrappers which check a thread-local retry counter (line 2) to choose whether to execute the fallback or the transaction. First, the counter is incremented, line 4, and the transaction begins, line 8. If the transaction aborts, the execution is resumed in line 4 and the retry counter is incremented again. Once the number of retries is greater than MAX_RETRIES, line 5, the global fallback lock is taken and the execution is serialized, line 6. The transaction must subscribe to the global lock by reading it, so that it aborts when other thread

```
 1: global fallbackLock ← 0              ▷ Global lock for fallback execution
 2: thread local retries ← 0             ▷ Retry counter for transactions
 3: procedure TX_START(retries)
 4:   retries ← retries + 1
 5:   if (retries > MAX_RETRIES) then
 6:     acquire(fallbackLock)            ▷ Fallback begin
 7:   else
 8:     xBegin(4)                        ▷ Go to line 4 on abort
 9:   end if
10: end procedure
11: procedure TX_STOP(retries)
12:   if (retries ≤ MAX_RETRIES) then
13:     xCommit()                        ▷ Transaction end
14:   else
15:     release(fallbackLock)            ▷ Fallback end
16:   end if
17:   retries ← 0                        ▷ Reset retry counter
18: end procedure
```

Fig. 2. Simplified API implementation for hardware transactions

```
 1: chunk ← N/#TH
 2: start ← tid*chunk
 3: stop ← MIN(N, start + chunk)
 4: for (t ← 0; t ≤ N-2; t ← t+1) do
 5:   for (k ← start; k < stop; k ← k+1) do
 6:     → TX_START(tx)                   ▷ Not necessary when using check_spec
 7:     if (k < (N-t-1)) then
 8:       W[t+k+1] ← W[t+k+1] + B[k][t+k+1]*W[t]
 9:     end if
10:     → TX_STOP(tx) or CHECK_SPEC(tx)  ▷ Speculation checkpoint
11:   end for
12:   BARRIER( ) → SPEC_BARRIER(tx)      ▷ Replace the barrier with an SB
13: end for                              ▷ to enable speculation
14: → LAST_BARRIER(tx)                   ▷ Non-speculative barrier
```

Fig. 3. Parallel Livermore loop 6 (Recurrence). Right arrows point to the changes needed for SB utilization (see Section 2)

runs the fallback code. Other optimizations [17] can be approached. At the end of the critical section the counter is checked again to see whether it is the transaction or the fallback executing, line 12. In the first case the transaction is committed, line 13. Otherwise, the fallback lock is released, line 15. Finally, the retry counter is reset in line 17.

1.2 Barriers: A motivational case study

As mentioned in the introduction, barrier-intensive application performance can be hindered by load imbalance and synchronization penalties [12], [18]. Next, we motivate speculative barriers with a case study in linear recurrence.

Recurrence is the sixth kernel of the Livermore Loops (LFK) [19]. It implements a general recurrence equation over an array W of length N. The code for Recurrence, in Fig. 3, first distributes the computation among threads in lines 1 to 3. The inner loop in lines 5 to 11 updates a single element of W at a time by accessing preceding elements of W and matrix B. This creates a recurrence due to read-after-write (RAW) patterns in W, so the barrier in line 12 is mandatory at the end of each iteration of the outer loop (lines 4 to 13).

Fig. 4 shows the execution of the first two iterations of the outer loop with N=8 and #TH=2. The thread on the left, th0, updates elements 1 to 4 of W while th1 updates elements 5 to 7. After the barrier, th0 updates elements 2 to 5, with a RAW dependence in 5, and th1 updates elements 6 to 7, with a RAW dependence in element 1 which is used for the W updates in line 8. Cross-barrier dependencies in Recurrence

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

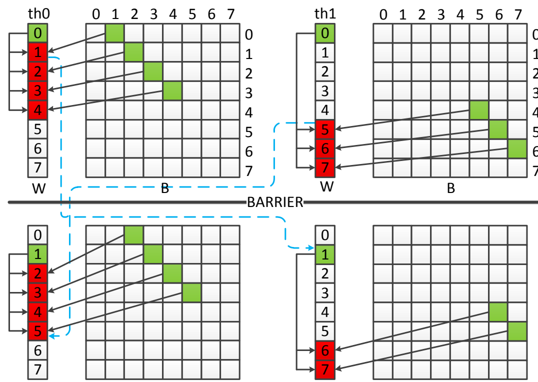

Fig. 4. Cross-barrier dependencies for Recurrence. Green squares represent reads, red squares represent writes, dashed arrows represent cross-barrier dependencies.

are low, but the barrier is needed to ensure correctness, with the consequent communication latency and load imbalance penalties. This cross-barrier dependency pattern might lead to various scenarios where SBs can extract performance:

- th0 finishes the first iteration of the outer loop before th1 and crosses the barrier speculatively. By the time th0 gets to compute element 5 of W in the second iteration, th1 already computed element 5 and th0 can safely read it without risking abort[1]. The fact that the element holding the dependency is the first to be computed by th1 in the first iteration and the last to be computed by th0 in the second iteration makes this scenario more feasible.
- th1 finishes the first iteration of the outer loop before th0 and crosses the barrier speculatively. When th1 computes element 6 of W in the second iteration, th0 already computed element 1 and th1 can safely consume it. The first element to be computed by th1 in the second iteration depends on the first element to be computed by th0 in the first iteration. It is less distance than that of the first scenario but can be greater depending on the chunk size (line 1, Fig. 3).

We evaluated our proposals with seven barrier-intensive benchmarks. Five present dependency patterns and load imbalance similar to Recurrence. The rest lacks these characteristics, with the purpose of checking SBs overhead. See Section 3.2 and Appendix A for more information.

2 Speculative Barriers

SBs [14] are intended to be used in transactional codes where transactions can be found between barriers. In outline, an SB works as follows. A thread elides the barrier and starts a transaction for after-barrier speculation. We call this transaction the *SB transaction*. If a transaction is found after the barrier, the *xBegin()* instruction is not executed and its *xCommit()* is replaced with a check to end the SB transaction. The speculation will continue until (i) a conflict is detected, (ii) the speculation limit is reached or (iii) all other threads crossed the barrier. Case (i) will abort the SB transaction, discard speculative work, and retry the SB transaction. In (ii), the SB transaction will wait for the other threads to cross the barrier. Thus, speculation is limited as best-effort HTM systems have limited resources. In (iii), the SB transaction will commit since the speculation was successful.

2.1 API implementation

Fig. 5 provides a detailed API implementation for SBs. We define two global variables: *sbBarrier*, in line 1, to keep count of the number of threads that crossed the SB; and *sbOrder*, in line 2, which maintains the global order of the current non-speculative barrier and is used to implement the partial order restrictions among transactions.

Each thread has a local transaction descriptor, lines 3 to 9, to implement the SB functionality. It includes a local *order* variable (line 4) that is incremented each time the thread reaches an SB. This way, a thread is considered speculative if its local *order* is ahead of *sbOrder*. A *spec* flag (line 6) represents the speculative state of the corresponding thread. *sbOrder* is incremented once the last thread reaches an SB (line 68), enabling the speculative threads to commit.

TX_START, line 11: this procedure replaces the HTM primitive to begin a transaction (*xBegin()*). It first checks if the thread is speculating, i.e. the thread elided a barrier and started an SB transaction, in which case nothing is done since it is already executing a transaction. If not, the standard procedure to begin a transaction is executed (see Fig. 2). An eager subscription fallback mechanism [17] is engaged after a MAX_RETRIES number of retries.

TX_STOP, line 21: this procedure replaces the HTM primitive to commit a transaction (*xCommit()*). In this case, we consider two different situations. If the corresponding thread is not speculating (lines 22 to 30), the standard procedure to commit a transaction is executed (see Fig. 2) and the corresponding variables of the descriptor are reset. In case of speculation, we can either end the speculation or continue. In the first case, we check if the orders match (line 32). Note that the order check must be escaped (line 31). Otherwise the SB transaction would be aborted by a *sbOrder* update. In the second case, the SB transaction continues unless the maximum speculation level is reached, in which case the SB transaction is blocked until the orders match or the transaction is aborted.

SPEC_BARRIER, line 54: this procedure is intended as a substitute for standard barriers. Any non-speculative thread that reaches an SB increments its local *order* (line 65). The last thread reaching the SB also increments *sbOrder* to generate a new global order (line 68). If the thread is not the last, it would be blocked in a traditional barrier, but here it elides the barrier and begins a new SB transaction instead (line 83). A subsequent abort of such transaction will return the execution to line 70, where the thread checks whether the speculation must end (lines 71 to 74) or be adjusted otherwise (lines 76 to 81). The last thread does not initiate an SB transaction since there is no need to speculate.

Speculation in our proposal is limited to one SB per thread as resources are limited in a best-effort HTM system. Hence, any speculative thread that reaches a subsequent SB waits until its local *order* matches *sbOrder*. This is done

1. If th0 computes element 5 in the second iteration before th1 does in the first iteration, the speculative after-barrier transaction opened by th0 will be aborted by the th1 access to element 5 due to strong isolation.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers


```
1: global sbBarrier ← #TH                          ▷ Barrier counter
2: global sbOrder ← 1                              ▷ Global order for barriers
3: thread local tx {                               ▷ Transaction descriptor
4:     order ← 1                                   ▷ Local order for the transaction
5:     retries ← 0                                 ▷ Retry count for the transaction
6:     spec ← 0                                    ▷ Flag to signal SB mode
7:     specMax ← MAX_SPEC                          ▷ Maximum spec level
8:     specLevel ← MAX_SPEC                        ▷ Current spec level
9: }
10:
11: procedure TX_START(tx)
12:     if (tx.spec = 0) then                      ▷ Not executing an SB
13:         tx.retries ← tx.retries + 1
14:         if (tx.retries > MAX_RETRIES) then
15:             fallbackBegin()                    ▷ Acquire fallback lock
16:         else
17:             xBegin(13)                         ▷ Go to line 13 on abort
18:         end if
19:     end if                                     ▷ If the thread is in SB mode, do nothing
20: end procedure
21: procedure TX_STOP(tx)
22:     if (tx.spec = 0) then                      ▷ Not in SB mode
23:         if (tx.retries ≤ MAX_RETRIES) then
24:             xCommit()
25:         else
26:             fallbackEnd()                      ▷ Release fallback lock
27:         end if
28:         tx.retries ← 0                         ▷ Reset tx descriptor
29:         tx.specLevel ← tx.specMax
30:     else                                       ▷ In SB mode
31:         escapeBegin()                          ▷ Suspend HTM system tracking
32:         if (tx.order ≤ sbOrder) then
33:             escapeEnd()                        ▷ Resume HTM system tracking
34:             xCommit()
35:             tx.retries ← 0                     ▷ Reset tx descriptor
36:             tx.specLevel ← tx.specMax
37:             tx.spec ← 0
38:         else
39:             escapeEnd()                        ▷ Resume HTM system tracking
40:             tx.specLevel ← tx.specLevel - 1;
41:             if (tx.specLevel = 0) then
42:                 escapeBegin()                  ▷ Suspend HTM system tracking
43:                 while (tx.order > sbOrder) do
44:                 end while                      ▷ Busy waiting
45:                 escapeEnd()                    ▷ Resume HTM system tracking
46:                 xCommit()
47:                 tx.retries ← 0                 ▷ Reset tx descriptor
48:                 tx.specLevel ← tx.specMax
49:                 tx.spec ← 0
50:             end if
51:         end if
52:     end if
53: end procedure
54: procedure SPEC_BARRIER(tx)
55:     if (tx.spec = 1) then                      ▷ Executing an SB
56:         escapeBegin()                          ▷ Suspend HTM system tracking
57:         while (tx.order > sbOrder) do
58:         end while                              ▷ Busy waiting
59:         escapeEnd()                            ▷ Resume HTM system tracking
60:         xCommit()
61:         tx.retries ← 0                         ▷ Reset tx descriptor
62:         tx.specLevel ← tx.specMax
63:         tx.spec ← 0
64:     end if
65:     tx.order ← tx.order + 1;
66:     if ((sbBarrier ← sbBarrier - 1) = 0) then  ▷ Atomic
67:         sbBarrier ← #TH
68:         sbOrder ← sbOrder + 1                  ▷ Atomic
69:     else
70:         tx.retries ← tx.retries + 1
71:         if (tx.order ≤ sbOrder) then           ▷ Do not begin an SB
72:             tx.retries ← 0                     ▷ Reset tx descriptor
73:             tx.specLevel ← tx.specMax
74:             tx.spec ← 0
75:         else
76:             if (tx.retries > MAX_RETRIES) then
77:                 if tx.specMax > 1 then
78:                     tx.specMax ← tx.specMax - 1
79:                 end if
80:                 tx.specLevel ← tx.specMax
81:             end if
82:             tx.spec ← 1
83:             xBegin(70)                         ▷ Go to line 70 on abort
84:         end if
85:     end if
86: end procedure
87: procedure LAST_BARRIER(tx)
88:     if (tx.spec = 1) then                      ▷ Executing an SB
89:         escapeBegin()                          ▷ Suspend HTM system tracking
90:         while (tx.order > sbOrder) do
91:         end while                              ▷ Busy waiting
92:         escapeEnd()                            ▷ Resume HTM system tracking
93:         xCommit()
94:         tx.retries ← 0                         ▷ Reset tx descriptor
95:         tx.specLevel ← tx.specMax
96:         tx.spec ← 0
97:     end if
98:     tx.order ← tx.order + 1;
99:     if ((sbBarrier ← sbBarrier - 1) = 0) then  ▷ Atomic
100:        sbBarrier ← #TH
101:        sbOrder ← sbOrder + 1                   ▷ Atomic
102:    else
103:        while (tx.order > sbOrder) do
104:        end while                               ▷ Busy waiting
105:    end if
106: end procedure
```

Fig. 5. API implementation for Speculative Barriers (SBs)

by an escaped busy-waiting to avoid the abort of the SB transaction due to the access to *sbOrder* (lines 56 to 59). Once the orders match, the SB transaction is committed and the descriptor is reset.

LAST_BARRIER, line 87: this procedure is similar to SPEC_BARRIER, but no speculation is allowed afterwards. This barrier should be used if we are certain that there is no possibility of speculation after the barrier.

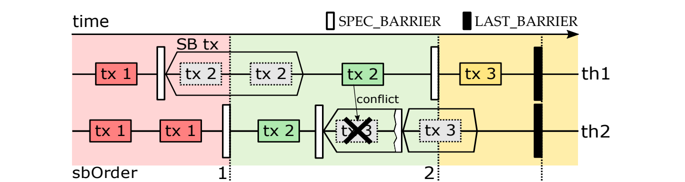

Fig. 6. SB scenario.

Scenario: Fig. 6 shows a scenario using SBs. Dotted lines represent each phase of the computation. First, th1 and th2 execute transactions (tx #, with # being the local order of the transaction) before the first barrier. The *spec* flag is set to 0 so transactions begin and commit normally. Then, th1 reaches an SB, increments its local order, decrements *sbBarrier* and finds it is not the last. Consequently, th1 begins an SB transaction and continues its execution. By the time the second thread reaches the barrier, th1 has already executed one transaction, which is not actually opened due to the SB transaction, hence the dotted rectangle. However, th1 cannot commit the SB transaction until the end of the second transaction because the global order check in the commit of the first one was performed too early (its local order was greater than the global order). Next, th2 executes one

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

transaction and elides another barrier by opening an SB transaction. The SB transaction is aborted and retried due to a conflict with a standard transaction from th1. We can see how threads cross the SBs without waiting each other.

2.2 SB-aware conflict resolution enhancement

A conflict between a standard transaction and an SB transaction is shown in the scenario of Fig. 6. In this example the SB transaction in th2 is aborted by the standard transaction in th1, but the conflict could have been resolved the other way around. If so, an SB transaction would delay other threads crossing the barrier, and the SB transaction commit would be delayed consequently. This situation may cause a performance loss in systems with a traditional (requester-wins or requester-loses) conflict resolution policy. For that reason we propose an SB-aware conflict resolution enhancement so that the SB transaction always stalls on such conflicts. In fact, our *SB-stalls* policy uses the requester-wins policy among standard transactions, and among SB transactions, but switches to SB-stalls for conflicts involving SB transactions and standard ones.

To implement the SB-stalls policy we need a new *SB-tx* bit per hardware context to flag whether the ongoing transaction is an SB transaction. A new instruction is also needed to set the SB-tx bit at the beginning of an SB transaction. The bit is implicitly reset in the commit. Besides, the SB-tx bit must be attached to coherence request messages for the local cache controller to have the information of the remote core to resolve the conflict, since the HTM conflict manager is implemented in cache controllers. Hence, the cache coherence protocol must be modified to manage NACKing (negative acknowledgement) depending on the SB-tx bit.

Table 1 shows the changes to be made to the L1 cache coherence protocol of a chip multiprocessor (CMP) highlighted in gray. We consider a baseline system such as the one described in Section 3.1. In summary, each tile of the CMP has a private L1 cache and one bank of a shared L2 cache. Cache controllers feature request/response queues to communicate through the network interconnect, and a mandatory queue which holds the requests from the CPU. L1 caches hold a pair of read and write transactional bits per block, R-tx and W-tx. The cache coherence protocol is MESI with a directory holding a full bit vector of sharers. Consequently, we can have block invalidation (INV) messages coming from the directory, due to block evictions and the inclusion property, for instance. As we can see in the first column of the table, if an INV message is received for a transactional block, (R-tx$\vee$W-tx), the transaction is aborted whenever (SB-tx$\vee\neg$remote(SB-tx)) is fulfilled, i.e. either we are in SB mode or the remote requesting core is not. In this manner, requester-wins is always enforced except for the case in the second column. That is, if we are not in SB mode and we receive an INV message from a remote core in SB mode, the remote core is NACKed. When the remote controller receives the NACK (fifth column) its mandatory queue is recycled, causing the conflicting request to be reissued until either the transaction commits or the remote transaction is aborted. Lastly, GetShared (GETS) and GetModified (GETM) messages from other cores can be forwarded to the controller. GETS messages to S blocks are served by the L2 controller, but GETS and GETM requests to E/M blocks are forwarded to L1 caches to check for conflicts. For such forwarded requests the modifications to the protocol are the same as for the INV requests. We have adopted the LogTM [8] requester-stalls conflict resolution policy for our SB-stalls protocol.

```
 1: procedure CHECK_SPEC(tx)
 2:   if (tx.spec = 1) then                          ▷ In SB mode
 3:     escapeBegin()                  ▷ Suspend HTM system tracking
 4:     if (tx.order ≤ sbOrder) then
 5:       escapeEnd()                   ▷ Resume HTM system tracking
 6:       xCommit()
 7:       tx.retries ← 0                        ▷ Reset tx descriptor
 8:       tx.specLevel ← tx.specMax
 9:       tx.spec ← 0
10:     else
11:       escapeEnd()                   ▷ Resume HTM system tracking
12:       tx.specLevel ← tx.specLevel - 1;
13:       if (tx.specLevel = 0) then
14:         escapeBegin()              ▷ Suspend HTM system tracking
15:         while (tx.order > sbOrder) do
16:         end while                               ▷ Busy waiting
17:         escapeEnd()                 ▷ Resume HTM system tracking
18:         xCommit()
19:         tx.retries ← 0                      ▷ Reset tx descriptor
20:         tx.specLevel ← tx.specMax
21:         tx.spec ← 0
22:       end if
23:     end if
24:   end if
25: end procedure
```

Fig. 7. Procedure to check the SB transaction state.

2.3 Speculative barriers and non-transactional codes

The proposed API can be used *as is* with non-transactional codes. However, SB transactions might end up being too large for a HTM system to deal with if barriers are far away from each other. For example, Recurrence without standard transactions would end up with an SB transaction of size proportional to *chunk* (see Fig. 3). With the parameters from Table 2 and 2 threads, the size of the SB transaction would be up to 65K cache lines, which is huge for a commercial HTM system, resulting in capacity aborts. That is, not a single SB transaction would ever commit. Whereas in the presence of standard transactions between barriers we use the TX_STOP procedure to check the global order and potentially commit the SB transaction, in the absence of them we do not have such an intermediate checkpoint. Hence, we propose the CHECK_SPEC procedure.

Fig. 7 outlines the procedure, which is in essence the else clause of TX_STOP. Line 2 checks whether the thread is executing an SB transaction, and we try to commit the SB transaction if so. To this end, we check whether its local order is less than or equal to the global order (*sbOrder*) and commit the SB transaction. If not, the SB transaction goes on. Note that we keep the *specLevel* check in line 13 regardless of the absence of transactions between barriers, so we can control the length of the speculation. The CHECK_SPEC procedure can be also used in codes with transactions in case the code in between transactions is too long.

2.4 Speculative barrier use guidelines

Appendix A outlines the code of the benchmarks we used for evaluation. In the majority of the benchmarks the use

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

TABLE 1
L1 cache coherence protocol modifications to implement the SB-stalls policy.

State	Events				
	INV (R-tx∨W-tx)∧ (SB-tx∨¬remote(SB-tx)[1])	INV (R-tx∨W-tx)∧ (¬SB-tx∧remote(SB-tx))	FWD GET (R-tx∨W-tx)∧ (SB-tx∨¬remote(SB-tx))	FWD GET (R-tx∨W-tx)∧ (¬SB-tx∧remote(SB-tx))	NACK
I	–	–	–	–	z[2]
S	Abort /I	NACK	–	–	z
E	Abort /I	NACK	Abort /I	NACK	–
M	Abort /I	NACK	Abort /I	NACK	–

[1] *remote(SB-tx)*: gets the SB-tx bit from the requester, which is in the request message.
[2] *z*: recycles mandatory queue.

is as straightforward as replacing the barriers by SBs (see Fig. 3). If the code has transactions, the wrappers in Fig. 2 must be replaced by the ones in our API (Fig. 5). If not, we should place a CHECK_SPEC to prevent large SB transactions. Next, we summarize the steps to follow for easy and efficient SB API application.

We should examine one barrier at a time and do not replace the barrier if: (i) The code before the barrier allocates memory that is used after the barrier. Accessing non-allocated memory in SB mode may risk segmentation fault or transaction abort, depending on the HTM system; (ii) The code after the barrier executes instructions that always abort, (e.g. system calls or interrupts). If an SB is placed in this case we only risk performance penalties, if any; (iii) There are easy to detect dependencies which would prevent the speculation to succeed.

We should replace the barrier with an SB if: (i) Dependencies are not evident and the barrier is placed just in case; (ii) Dependencies vary depending on the data, e.g. algorithms with dependencies only in the boundaries of data partitions.

In case of barriers being far away each other, CHECK_SPEC calls can be placed to add more checkpoints. Fewer checkpoints will lead to larger SB transactions that can increase aborts due to capacity and cross-barrier dependencies. More checkpoints will produce smaller SB transactions reducing the speculation window. Our proposal uses *specLevel* and *specMax* (see Fig. 5) to decrease the speculation level at run time (in terms of number of checkpoints) if an SB transaction runs out of retries. Other approaches to release the user from the task of placing CHECK_SPECs could be explored (e.g. the compiler could implicitly place checkpoints based on static transactional information at compile time or based on dynamic data –hardware counters measuring the transactions size, for example– at run time).

3 Experimental Evaluation

3.1 Targeted architectures

RubyHTM

We modeled a best-effort HTM system with the goal of implementing the conflict resolution enhancement described in Section 2.2. We used Simics [20], a full system simulator that can be augmented with third-party profiling and modeling modules, such as Wisconsin GEMS's Ruby [15]. We modified Ruby, which is a multiprocessor memory system timing simulator. We installed an unmodified instance of Solaris 10 in an image of a SPARC CMP for Simics.

Our system, called RubyHTM, has 64 in-order single-issue cores with a private 64KiB 8-way L1D cache, where each L1 cache block has a pair of transactional read/write bits (R-tx and W-tx) which can be flash-cleared on transaction commit/abort; a unified, shared L2 cache divided into 64 banks of 1MiB of capacity, 16 ways and 64B blocks; a full bit vector of sharers directory; and a packet-switched crossbar interconnect. The HTM system baseline conflict resolution policy is requester-wins and we implemented our enhanced conflict resolution policy as well. Strong isolation [16] is enforced. The system supports escape actions.

IBM Power8

The Power8 processor includes best-effort HTM extensions [11] providing strong isolation, lazy version management, eager conflict detection with cache-line granularity, and a mixed conflict resolution policy with a requester-loses scheme for transactional loads and a requester-wins scheme for transactional updates. The design allows up to 8KB of transactional accesses per core. Escape actions are supported by a *suspended-mode* which can be enabled by using *tsuspend*/*tresume* instructions to delimit a non-transactional section inside a transaction. Experiments were carried out in a S822LC-8335 server with 2-socket, 10-core processors. Each core can execute up to 8 simultaneous threads (SMT) resulting in a total of 160 execution threads. Each core have the following characteristics: a 64 KiB L1 private write-through data cache; a 512 KiB L2 private data cache; 8 MiB of a semi-private L3 cache.

3.2 Benchmarks

We evaluated our proposals with seven barrier-intensive benchmarks. Table 2 summarizes their parameters and characteristics. The Barrier microbenchmark [21] represents a best-case scenario with no cross-barrier dependencies. Half of the threads work in each iteration creating a load imbalance. It was used to check the feasibility of the proposed SB API. Cholesky and Recurrence correspond with the second and sixth kernels of Livermore Loops (LFK) [19]. These kernels exploit fine-grain data parallelism with a high frequency of barriers [12]. Recurrence, introduced in Section 1.2, exhibits a dependency pattern and load imbalance suitable for SBs. Cholesky, based on the incomplete Cholesky factorization implementation in [12], has similar characteristics. We modified both Recurrence and Cholesky to add a resizable computation chunk, configurable by parameter C, to control transaction size. The number of barriers in these benchmarks depends directly on parameter

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

TABLE 2
Benchmark Parameters and Transaction and Speculative Commit Rate for 16 threads.

	Power 8			TCR (%)/SCR (%)			RubyHTM	TCR (%)/SCR (%)			
	Parameters	**#Barr**	**$\vert tx \vert$ (M/A)**	TM	SB	CS	**Parameters**	TM	SB	SB-stalls	CS
Barr	N100K L1K	100K	5/4.5	99.9/-	99.9/99.9	-/99.9	-	-	-	-	-
	N100K L10K	100K	5/4.5	99.9/-	99.7/99.9	-/99.9	-	-	-	-	-
Recurren	N20K C1	20K	9/6.5	98.6/-	99.3/6.3	-/34.6	N2K C1	98.8/-	98.8/48.9	98.8/47.9	-/50.7
	N20K C5	20K	16/8.9	95.5/-	98.6/8.3	-/33.1	N2K C5	88.0/-	86.6/21.6	87.7/31.7	-/49.0
	N20K C10	20K	21/11.8	92.2/-	97.9/8.4	-/26.7	N2K C10	81.9/-	79.5/16.7	82.0/26.4	-/47.9
	N20K C15	20K	26/14.8	88.6/-	96.9/7.9	-/29.2	N2K C15	79.8/-	76.4/14.5	80.4/22.9	-/45.0
Cholesky	N100M C1	27	8/7.2	100/-	100/38.0	-/60.9	N100K C1	99.9/-	100.0/86.1	100/81.4	-/84.4
	N100M C5	27	10/8.5	99.9/-	100/43.8	-/58.8	N100K C5	98.9/-	99.5/56.2	99.5/64.1	-/76.3
	N100M C10	27	12/10	99.9/-	100/45.1	-/59.3	N100K C10	97.7/-	98.8/51.9	98.8/58.2	-/74.9
	N100M C15	27	12/11.6	99.9/-	100/45.7	-/57.1	N100K C15	96.5/-	98.4/45.1	98.9/54.8	-/74.4
DGCA	N2K A2 T4K	4K	20/18.1	99.8/-	99.9/99.6	-/99	N500 A2 T500	85.1/-	84.2/2.7	91.2/5.1	-/18.2
	N8K A2 T4K	4K	20/18.1	99.9/-	99.9/99.4	-/99.7	N1K A2 T500	91.8/-	90.7/2.1	95.4/2.9	-/12.2
	N2K A8 T4K	4K	27/25.1	94.2/-	93.1/42.1	-/48.6	N500 A8 T500	77.1/-	74.7/2.7	77.6/6.8	-/6.4
	N8K A8 T4K	4K	26/24.6	98.1/-	97.8/29.6	-/40.3	N1K A8 T500	86.3/-	84.0/1.1	86.5/5.8	-/3.6
Sten	Large T100	101	12/11.1	99.9/-	99.9/10.3	-/25.5	Small T50	100/-	100/33.7	100/35.2	-/31.7
	Large T800	801	12/11.1	99.9/-	99.9/7.6	-/26.7	Small T100	100/-	100/37.9	100/38.7	-/33.8
Hist	Large T1K C1	2001	6/5.9	98.9/-	98.9/0.9	-/1.2	Small T20 C1	90.1/-	90.5/41.4	90.5/42.3	-/29.8
	Large T1K C5	2001	15/13.2	30.4/-	30.9/0.2	-/0.2	Small T20 C5	3.7/-	3.7/0.4	3.7/0.4	-/12.4
SSCA2	k1 -s14-l9-p9	10	8/8	98.6/-	98.5/3	-	k1 -s13-l3-p3	84.7/-	85.2/33.1	85.2/32.3	-
	k1 -s20-l3-p3	10	8/8	99.9/-	99.9/0.2	-	k1 -s14-l9-p9	91.9/-	91.9/47.1	91.9/45.5	-
	k4 -s14-l9-p9	24K	6/6	71.77/-	30.07/0.51	-	k4 -s13-l3-p3	33.5/-	23.1/0.09	23.5/0.09	-
	k4 -s20-l3-p3	1572K	6/6	62.61/-	30.47/0.05	-	k4 -s14-l9-p9	33.6/-	23.1/0.07	23.3/0.08	-

N, being $log_2(N)$ for Cholesky. DGCA [22] is a coloring graph algorithm with the number of nodes given by N and the node associativity by A. The number of barriers depends on T, the number of refining iterations of the algorithm. Stencil and Histogram belong to the Parboil suite [23], being a Jacobi and a saturating histogram application, respectively. We modified the OpenMP versions of them to got a pthread alternative for the kernels where the *omp parallel for* implicit barrier was replaced by an SB. We added a resizable chunk (C) to Histogram and computed different number of iterations (T), grids for Stencil (Large: 521x512x64, Small: 128x32x8) and image sizes for Histogram (Large: 996x1040, Small: 498x520). SSCA2 [24] is a graph theory application consisting of 4 kernels from which we used the first and the fourth ones. Kernel 1 does not exhibit many barriers, whereas kernel 4 shows plenty since they reside within a loop. However, the cross-barrier dependencies are very pronounced in this kernel.

We chose smaller data structure sizes and fewer iterations for RubyHTM due to the slow simulations. For SSCA2, we used the parameters proposed in [25] with their variants for simulation and real machines. Appendix A describes the code of the benchmarks and where we placed the SBs.

The $|tx|$ *(M/A)* column shows the maximum (M) and average (A) size of standard transactions measured in accesed cache blocks. We found that, with these sizes, and considering bookkeeping limitations on tested HTMs, a value of MAX_SPEC=4 worked well for the benchmarks evaluated.

3.3 Methodology

We obtained the average execution time of 20 executions of each experiment. More consistent results were obtained by bounding each thread to a processor to prevent migrations. In the Power8 machine we used up to 128 cores with threads' affinity set in a round robin scheme avoiding SMT when possible. This fact gives rise to three different scenarios to consider: (i) experiments using 1-8 threads are executed in a single socket, so the impact of communication among different cores is reduced. Also, a single physical core is used per thread, so the thread can use all the transactional and hardware resources available. (ii) Experiments with 16 threads are executed in 2 sockets, with the communication penalty that it implies, but the threads still use a single physical core per thread. (iii) Experiments with 32 to 128 threads are executed in 2 sockets with SMT, sharing transactional and hardware resources and suffering the inter-socket communication latency.

We used *padding* to ensure that variables in lines 4 and 5 of Fig. 5 are mapped to separate cache blocks, thus preventing false sharing conflicts. The counter *tx.order* is read in escape action and may self-abort the transaction as other descriptor variables are written inside the transaction.

Regarding the performance results we considered six different profiles:

- *Unprotected*: neither transactions nor barriers were used, so the results may be incorrect. This profile represents a performance upper limit.
- *Parallel*: features standard barriers already present in the code and neither transactions nor SBs are used.
- *CS*: similar to Parallel, but barriers are replaced with SBs, and the CHECK_SPEC primitive is used to add additional checkpoints to the speculation.
- *TM*: features transactions throughout the code and standard barriers already present in the benchmarks.
- *SB*: similar to TM, but barriers are replaced with SBs. In this case, transactions after the barrier already provide checkpoints to end the speculation, so the CHECK_SPEC primitive is not used.
- *SB-stalls*: same as SB, but with the enhanced coherence protocol described in Section 2.2. This profile is only present in the simulator experiments.

Metrics analyzed include *speedup* with respect to non-transactional, sequential execution; *transaction commit rate* (TCR), which is the percentage of committed transactions

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

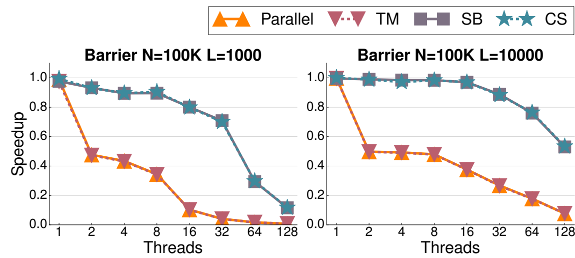

Fig. 8. Barrier microbenchmark results using Power8.

with respect to all the transactions initiated, and *speculative commit rate* (SCR), which is the percentage of committed SB transactions with respect to all the SB transactions initiated.

3.4 Results

Results for all the benchmarks are shown in Fig. 8 (Barrier), Fig. 9 (Power8), Fig. 10 (RubyHTM) and Table 2 (SCR/TCR).

Barrier microbenchmark

In this benchmark the Y-axis represents efficiency rather than speedup, as it is implemented so that the load is not shared among threads but replicated. This way we can measure the time wasted by transactions and barriers.

Due to the load imbalance among iterations (N), the efficiency in profiles without speculation (Parallel and TM) is halved when using more than one thread. In these cases, only half of the threads perform actual work in each iteration of the inner loop, L (see code in Appendix A). However, SB and CS profiles can speculate upon reaching the barrier, so the idle threads in a given iteration can immediately advance and work in the next one. On the left, we have reduced the load (L) of each iteration to observe the barrier contention. Parallel and TM profiles are penalized when using more threads due to both the barrier contention and the load imbalance. SB and CS profiles also decrease their performance because of the contention, but the penalty is reduced since the threads do not need to wait in barriers. All these experiments yield about 100% SCR with both SB and CS profiles, which confirms that the proposed API does not introduce additional aborts due to false-sharing.

Recurrence

Two situations can be observed in this benchmark: Unprotected, Parallel and CS profiles do not have the overhead of between-barrier transactions (e.g. memory barriers on beginning and committing), while TM and SB profiles do. Such differences can be clearly identified when using 1-8 threads, where Unprotected, Parallel and CS profiles obtain similar speedups, whereas TM and SB do it but to a much lower extent. Executions with 16 threads exhibit super-linear speedups with Unprotected and CS in Power8 due to cache interaction (see Appendix B for more information). Parallel and TM profiles do not scale due to the increased cost of synchronization when using 2 sockets, while the barrier speculation in the SB profile noticeably reduces the synchronization overhead. Results with 32 to 128 threads show the overhead of SMT resource sharing. Performance decreases for all profiles, with larger chunks being more affected due to the combination of larger transactional footprints and shared transactional resources.

RubyHTM results are comparable to Power8. This time, however, all threads execute in a single socket and transactional resources are not shared among threads, so the slowdowns observed in Power8 are not so profuse here. A slight super-linear behaviour is also obtained with the Unprotected and CS profiles with 8 threads (see Appendix B), and the additional SB-stalls profile obtains a performance boost compared with SB for 32 and 64 threads.

Regarding TCR, all profiles obtain good results in this benchmark. Larger chunk sizes produce slightly worse ratios, and SB-stalls yields better results than SB for both TCR and SCR. Nevertheless, best SCR are obtained with CS, which is expected, since there are less in-flight transactions to conflict with. Notice that Power8 obtains lower values for SCR than RubyHTM. Reasons for this behaviour include false sharing due to cache line granularity (doubled in Power8), differences in conflict resolution policy and smaller transactional cache in Power8.

Cholesky

Cholesky shows less dependencies than Recurrence, which is reflected in better TCR and SCR values in all scenarios.

Power8 performance using CS sits between Unprotected and Parallel regardless of the number of threads. Note that relative performance differences among such profiles increase from 16 threads onwards due to the higher communication costs when using 2 sockets. This behaviour is also present when comparing TM and SB profiles. In this case, however, differences are more evident with 32 to 128 threads. The overhead caused by transactions partially masks relative performance differences. The effect of resource sharing when using SMT can be noticed as well.

RubyHTM results show reduced performance differences among different profiles except for Unprotected, due to the smaller number of iterations (N). As in Recurrence, the effect of inter-socket communication and SMT is not present in the simulator. Differences between SB and SB-stalls are negligible due to the low conflict ratio and the reduced number of barriers.

DGCA

DGCA plots depict different configurations varying the node adjacency (A) and the node count (N) of the input graph. Individual connections between nodes are randomly chosen when constructing the graph, following a uniform distribution. This way, we explore low-contention scenarios with small and large graphs in the first two plots, and high-contention scenarios in the rest. As a consequence of the simulation overhead, RubyHTM experiments use smaller graphs and fewer iterations (M).

Power8 executions in low-contention scenarios exhibit a noticeable overhead due to transactions when using 1 to 8 threads. In this case, Unprotected and Parallel show a significant performance gap with respect to CS, while TM and SB profiles perform similarly. Nevertheless, Parallel is heavily affected by barrier synchronization from 8 threads onwards, obtaining speedups comparable to TM with high thread counts. Speculation performs well in these scenarios, with CS and SB scaling up to 32 threads and obtaining

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

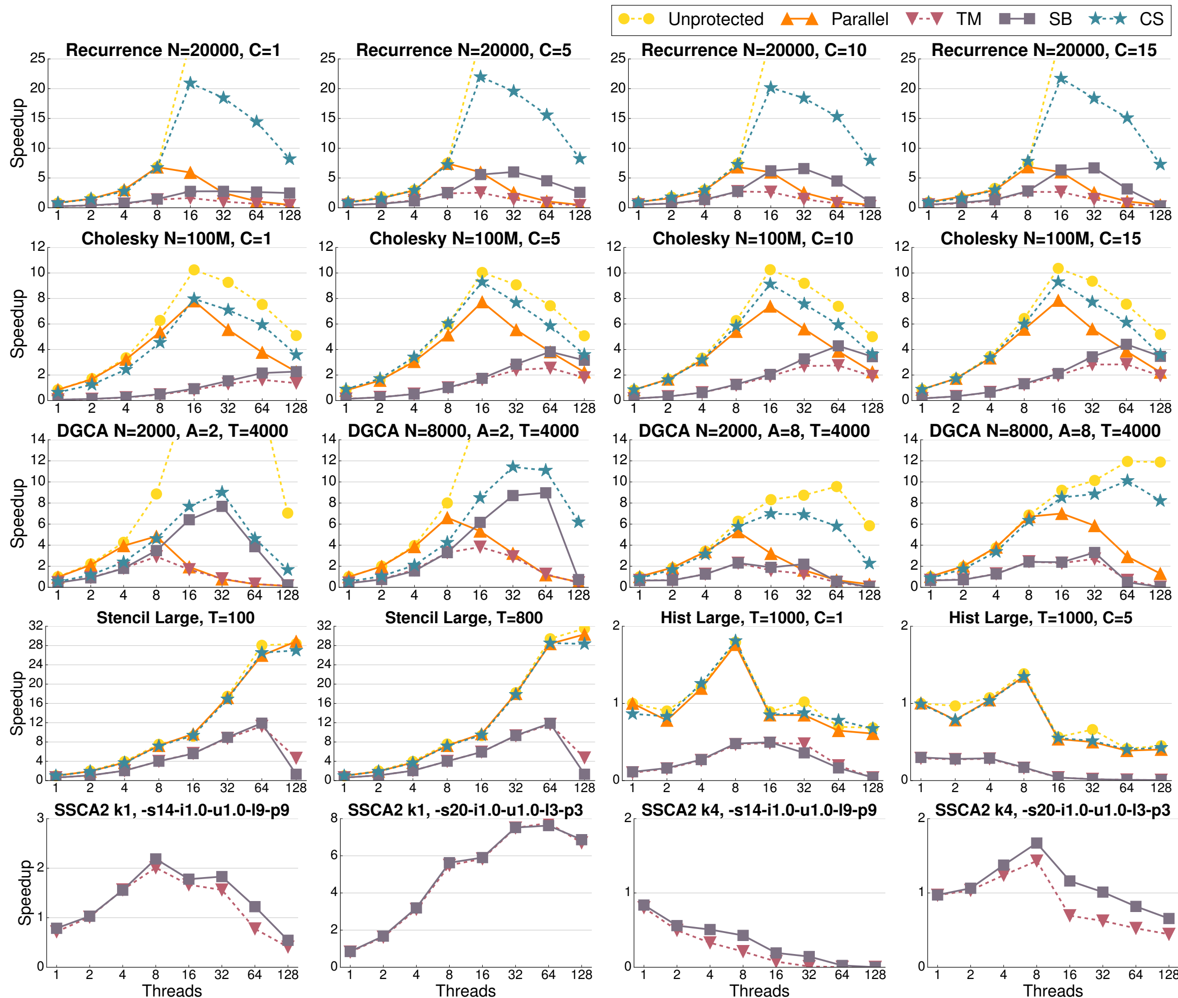

Fig. 9. Power8 results for Recurrence (Livermore loop 6), Cholesky (Livermore loop 2), DGCA, Stencil, Histogram and SSCA2 (Kernel 1 and 4).

the highest SCR values. The larger graph (N=8000) in the second plot shows a similar behaviour, and extends the performance gains with speculative profiles to 64 threads, because of the lower conflict probability.

Scalability in high-contention scenarios is limited mainly by dependencies. In these experiments the differences with 1 to 8 threads among CS, Parallel and Unprotected are reduced due to more compute-intensive executions associated with higher node connectivity. The same behaviour is observed between SB and TM. It is worth noting in the rightmost plot the performance differences between CS and SB profiles from 32 threads onwards. In this case, SB cannot maintain the same scalability, which is expected since standard transactions share transactional resources with SB transactions. Using SMT threads gives rise to aborts caused by footprint overflow. This neither happens with CS, because non-speculative threads do not need such transactional resources, nor with A=2, where fewer transactional accesses are needed due to the lower connectivity.

RubyHTM yields a similar behaviour. TM, SB and SB-stalls profiles add the overhead of standard transactions, but the CS profile can leverage speculation for a larger number of threads. This is expected for two reasons: first, graph sizes are smaller in RubyHTM executions giving rise to more transactional conflicts compared with Power8. Second, the use of smaller M values in the simulator affects to the conflict probability, which decreases with each iteration of the corresponding loop as node colors are updated. Nevertheless, SB-stalls yields up to 33% performance gain with respect to SB in executions from 32 threads.

Differences in experiment configurations are specially evident in TCR and SCR results. Power8 exhibits high TCR in all experiments, while SCR depends on both the graph size and the number of connections, which determine the potential conflicts. RubyHTM figures, though, are lower due to the use of smaller graph sizes and fewer iterations. This is specially evident for SCR.

Stencil and Histogram

These benchmarks represent bad-case scenarios for two reasons: first, they do not exhibit load imbalance. Second, the barrier overhead is low, as we can see both in the number of barriers in Table 2 and in the small performance differences between Unprotected and Parallel profiles. Such differences hint the lack of opportunities for improvement with SBs.

Stencil exhibits cross-barrier dependencies only in the boundaries of the input data grid partitions. Speculative profiles show good values for TCR and moderate SCRs similar to that of Recurrence. However, such speculation is not translated into performance due to the reasons above.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

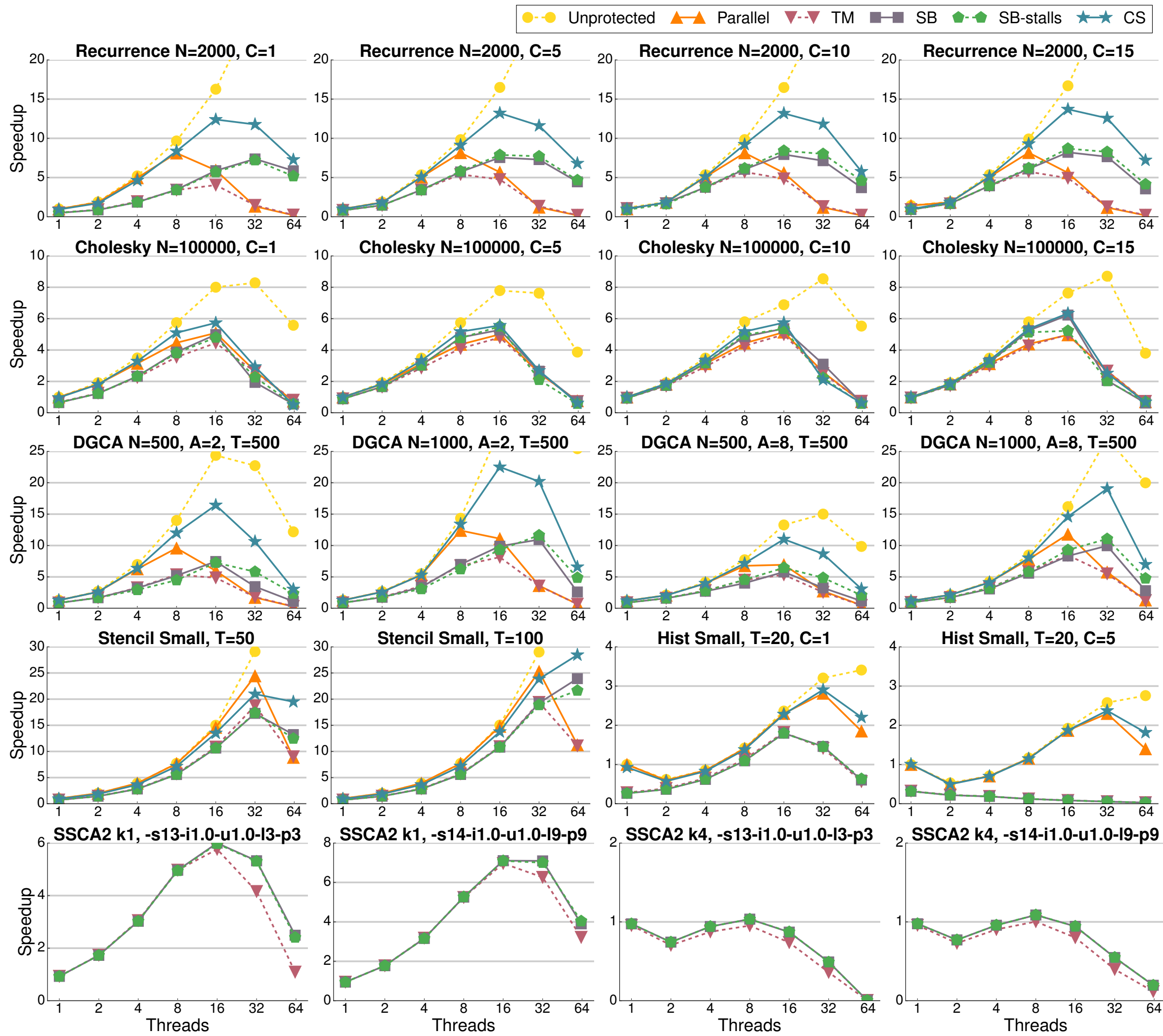

Fig. 10. RubyHTM results for Recurrence (Livermore loop 6), Cholesky (Livermore loop 2), DGCA, Stencil, Histogram and SSCA2 (Kernel 1 and 4).

Histogram has hard cross-barrier dependencies, reflected in the low SCR values except for RubyHTM (C1), which still does not leverage the speculation to increase the performance. Besides, the nature of the input, which follows a Gaussian pattern, causes multiple conflicts when updating the histogram, degrading TM and SB performance. This behaviour is exacerbated with larger chunks.

These benchmarks show that the performance of SBs in non-friendly scenarios is comparable to the use of traditional barriers, being the exception a slight performance decrease in Stencil when using 128 threads.

SSCA2

For SSCA2 we used kernels 1 and 4 with recommended inputs from [25]: small (s13) and medium (s14) inputs for RubyHTM, and medium (s14) and large (s20) for Power8. As part of STAMP, this benchmark is already parallelized using transactions, so only SB and TM profiles are considered.

Kernel 1 shows limited gains in Power8 with medium input parameters and no gains with large ones for SB over TM. Most of the barriers in this kernel protect dynamic allocations thus impeding the use of SB. Also, unlike previous benchmarks, SBs do not appear inside loops. As a consequence, only a few SBs are called throughout the code, reducing the speculation opportunities. RubyHTM results show better speedups than Power8 for Kernel 1, but a similar behaviour when comparing TM and SB profiles. The absence of inter-socket communication and SMT favors scalability. Limited gains are obtained with speculative profiles and a large number of threads.

Main speculation opportunities in Kernel 4 reside in the barriers within a loop which contains two SBs (see code in Appendix A). The iteration computation decreases as the loop advances, so some speculation can be leveraged in spite of the cross-barrier dependencies. Unfortunately, conflicts caused by a reduction operation and the frequent interactions with global variables cause aborts that limit performance. This translates into the small speedups in Fig. 9 and 10. Such interactions are specially harmful in Power8 due to its larger cache line size, which increases aborts due to false sharing. This fact can be observed in the large TCR differences (see Table 2). Regardless of the conflicts and low SCR, both Power8 and RubyHTM yield consistent performance gains of SB over TM. SB-stalls does not yield any benefit in SSCA2.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

Implications

The performance of SBs depends on several factors: the potential speculation opportunities can be seen as a combination of the number of barriers and load imbalance. The cross-barrier dependencies will determine the success ratio of such opportunities. Moreover, on real hardware we must account for the limited transactional resources, which limits speculation in terms of memory accesses. Consequently, SBs will perform better when there are many speculation opportunities offering little gains than the other way around.

Ideally, a high barrier count, irregular load and no cross-barrier conflicts represent the best scenario for SBs (Barrier benchmark). In our experience, load-imbalanced codes where the performance is fairly limited by barrier synchronization and cross-barrier dependencies offer certain window for speculation, representing good use cases (Recurrence, Choleksy, DGCA). Conversely, codes with a high ratio of cross-barriers dependencies (SSCA2 K4, Histo), well balanced or without enough barriers (SSCA K1, Histo, Stencil) will not extract significant performance gains.

SBs does not seem to harm performance with respect to traditional barriers in the vast majority of tested scenarios. Some losses can be observed on Stencil, where the overhead of traditional barriers is negligible. A runtime scheme could be adopted to avoid the use of SBs in such cases based on the stall-times spent on each barrier: If such times are shorter than a threshold, a LAST_BARRIER can be placed allowing traditional barriers to be used onwards. If stalls increase, this could be reverted to SBs. Some hysteresis may be beneficial for reducing both switching and time measurements.

SB-aware enhancement is beneficial when there are enough cross-barriers conflicts involving aborts of non-SB transactions due to the baseline HTM conflict resolution policy. Larger TCR differences between SB and SB-Stalls profiles hint situations where SB-stalls can extract performance.

4 Related Work

Martínez and Torrellas [26] propose hardware support for speculation in synchronization primitives including locks, flags and barriers. Ad hoc HTM-like hardware is added to caches and controllers to manage speculation. They do not support ordering among speculative threads and ensure progress only on deadlock-free parallel codes. To avoid unnecessary aborts due to barrier counter updates they expose the speculative hardware to the user. We provide barrier speculation with partial ordering, ensure forward progress due to HTM deadlock-free nature, and use escape actions to solve the barrier counter problem, while hiding it from the user with our API.

Porter et. al. [27] explore the path of efficient speculative multithreading in concert with a HTM subsystem. They execute a sequential application and spawn a speculative thread at run time whenever a loop iteration or a function is started. HTM supports the execution of the speculative thread. They show a 10% speedup in a simulated 2-core CMP and propose hardware optimizations such as data forwarding, ordered transactions or false sharing elimination with modest results. Our proposal is focused on barrier speculation for parallel codes, without new thread spawning, and supports ordered transaction commit without additional hardware requirements by leveraging escape actions. Our CS averages a $4\times$ speedup over Parallel for 128 threads.

Other approaches address thread-level speculation (TLS) with commercial HTM extensions. Odaira and Nakaike [28] analyse the limitations of Intel TSX when used to enable speculation on SPEC benchmarks. They focus on manually parallelizing frequently executed loops —not barriers— by enclosing iterations into transactions which are subsequently executed in parallel with order restrictions. As TSX does not support ordered transactions, speculative iterations completed out of order are aborted. Although some scenarios can leverage speculation, the performance is degraded in most cases. Interestingly, the main source of overhead is not the lack of ordered transactions, but transactional aborts due to conflicts.

Salamanca et al. [29] propose a TLS extension with commercial HTM support for OpenMP to ease TLS tuning and programming, and study the effect of false sharing.

Nagarajan and Gupta [18] analyse a set of parallel programs with fine grained barrier synchronization and propose a two-version code to gain performance by speculating on barriers. One version elides the barrier under the assumption that no interprocessor dependencies will arise. The other is the original traditional barrier version of the code. The former is speculatively executed first and the latter is executed in case of misspeculation. The compiler ensures speculative thread isolation by writing to a separate address space. They modify the Advance Load Address Table (ALAT) of Itanium processors to detect potential misspeculation at runtime. Their simulations show an average 12% reduction in execution time with 8 cores. Our proposal does not need compiler/hardware modifications nor an additional optimistic version of the code, and CS achieves a $4\times$ average speedup over Parallel with 128 threads.

In [21], after-barrier speculation using Intel Haswell HTM support is studied. The system does not support ordered transactions nor escape actions. Consequently, if speculative threads cannot commit upon reaching the speculation checkpoint, speculative work is discarded. For this reason, a best-case SCR of 40% is reported for the Barrier microbenchmark. The same benchmark yields nearly perfect SCR with our proposal due to the use of escape actions for communication among transactions (see Table 2). This work does not provide multiple speculative checkpoints nor adaptive speculation level as we do. Also it does not consider the limitations of commercial HTM systems including limited transactional sizes, presence of illegal instructions in speculative transactions, and conflict resolution policies.

5 Conclusions

In this paper we use commercial hardware transactional (HTM) extensions to implement an optimistic speculative barrier (SB) as an alternative to traditional barriers. SBs allow threads to elide a barrier by leveraging HTM. We provide an SB API implemented with HTM extensions to be used with transactional and non-transactional applications.

Our proposals are evaluated using a GEMS-based simulator architecture and an IBM Power8 server which provide escape actions, a hard requirement of SBs. Results show an overall performance improvement of SBs over traditional

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3044234, IEEE Transactions on Computers

barriers in benchmarks with load imbalance and low cross-barrier dependencies. Significant performance gains are obtained in Power8 as the number of threads increases. The 2-socket configuration penalizes traditional barrier communication and SBs help on hiding this latency. The overhead of SBs does not seem to affect performance negatively on the majority of the benchmarks tested.

ACKNOWLEDGMENTS

Governments of Spain (project TIN2016-80920-R) and Junta de Andalucía (project P12-TIC-1470) supported this work.

REFERENCES

[1] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Int'l. Symp. on Computer Architecture*, 1993, pp. 289–300.
[2] N. Shavit and D. Touitou, "Software transactional memory," in *Symp. on Principles of Distributed Computing*, 1995, pp. 204–213.
[3] T. Harris *et al.*, *Transactional Memory, 2nd edition*. Morgan & Claypool Publishers, 2010.
[4] D. Dice *et al.*, "Transactional locking II," in *Distributed Computing*, 2006, vol. 4167, pp. 194–208.
[5] P. Felber *et al.*, "Time-based software transactional memory," *Trans. on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.
[6] L. Hammond *et al.*, "Transactional memory coherence and consistency," in *Int'l. Symp. on Computer Architecture*, 2004, pp. 102–113.
[7] R. Rajwar *et al.*, "Virtualizing transactional memory," in *Int'l. Symp. on Computer Architecture*, 2005, pp. 494–505.
[8] K. Moore *et al.*, "LogTM: Log-based transactional memory," in *Int'l. Symp. on High-Performance Computer Architecture (HPCA'06)*, 2006, pp. 254–265.
[9] C. Kevin Shum *et al.*, "IBM zEC12: The third-generation high-frequency mainframe microprocessor," *Micro*, vol. 33, no. 2, pp. 38–47, 2013.
[10] P. Hammarlund *et al.*, "Haswell: The Fourth-Generation Intel Core Processor," *Micro*, vol. 34, no. 2, pp. 6–20, 2014.
[11] H. Q. Le *et al.*, "Transactional memory support in the IBM POWER8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 8:1–8:14, Jan 2015.
[12] J. Sampson *et al.*, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *Int'l. Symp. on Microarchitecture*, 2006, pp. 235–246.
[13] M. J. Moravan *et al.*, "Supporting Nested Transactional Memory in logTM," in *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 359–370.
[14] M. Pedrero *et al.*, "TMbarrier: Speculative Barriers Using Hardware Transactional Memory," in *Euromicro Int'l. Conf. on Parallel, Distributed and Network-Based Processing*, 2018, pp. 214–221.
[15] M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.
[16] M. M. K. Martin *et al.*, "Subtleties of Transactional Memory Atomicity Semantics," *Comput. Archit. Lett.*, vol. 5, no. 2, p. 17, 2006.
[17] R. Quislant *et al.*, "Insights into the Fallback Path of Best-Effort Hardware Transactional Memory Systems," in *Int'l. Conf. on Parallel Processing*, 2016, pp. 251–263.
[18] V. Nagarajan and R. Gupta, "Speculative Optimizations for Parallel Programs on Multicores," in *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2010, pp. 323–337.
[19] J. T. Feo, "An analysis of the computational and parallel complexity of the Livermore Loops," *Parallel Computing*, vol. 7, no. 2, pp. 163–185, 1988.
[20] P. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.
[21] L. Bonnichsen and A. Podobas, "Using Transactional Memory to Avoid Blocking in OpenMP Synchronization Directives," in *Int'l. Workshop on OpenMP*, 2015, pp. 149–161.
[22] K. R. Duffy *et al.*, "Complexity analysis of a decentralised graph colouring algorithm," *Information Processing Letters*, vol. 107, no. 2, pp. 60–63, 2008.
[23] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.
[24] D. A. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *Int'l. Conf. on High Performance Computing*, 2005, pp. 465–476.
[25] C. Minh *et al.*, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Int'l. Symp. on Workload Characterization*, 2008, pp. 35–46.
[26] J. F. Martínez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," *Oper. Syst. Rev.*, vol. 36, no. 5, pp. 18–29, 2002.
[27] L. Porter *et al.*, "Mapping out a path from hardware transactional memory to speculative multithreading," in *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2009, pp. 313–324.
[28] R. Odaira and T. Nakaike, "Thread-level speculation on off-the-self hardware transactional memory," in *Int'l. Symp. on Workload Characterization*, 2014, pp. 212–221.
[29] J. Salamanca *et al.*, "Using hardware-transactional-memory support to implement thread-level speculation," *Trans. on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 466–480, 2018.

Manuel Pedrero received the MSc degree in computer engineering from the University of Cordoba in 2012, the M.S. in Mechatronics in 2014, and the PhD in Computer Architecture in 2018, both from the University of Malaga. Currently, he is working as a software engineer. His research interests include high performance computing, with special regard to transacional memory.

Ricardo Quislant received the MSc degree in computer engineering from the University of Granada, and the PhD from the University of Malaga, Spain, in 2006 and 2012, respectively. Currently, he is working as a researcher in the Department of Computer Architecture at the University of Malaga. His main research interests include computer memory systems and high-performance computing, with special regard to transactional memory.

Eladio Gutierrez received the MSc and PhD degrees in telecommunication engineering both from the University of Malaga, Spain, in 1995 and 2001 respectively. Since 2003 he has been an associate professor in the Department of Computer Architecture at the University of Malaga. His research interests include parallel architectures and algorithms, automatic parallelization, engineering education and graphics processing units.

Emilio L. Zapata received the MSc degree in Physics from the University of Granada in 1978 and the PhD degree in Physics from the University of Santiago de Compostela, Spain, in 1983. Since 1991, he has been a full professor in the Department of Computer Architecture at the University of Málaga. His main research interests are numerical and audiovisual applications, high performance architectures and compilation techniques for parallel computers.

Oscar Plata received the MSc and PhD degrees in physics from the University of Santiago de Compostela, Spain, in 1985 and 1989, respectively. He started as an assistant professor in the University of Santiago de Compostela where he became an associate professor in 1990. Since 2002, he is a full professor at the University of Malaga, Spain. His research interests include high-performance computing and parallel architectures and algorithms.

0018-9340 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidad de Malaga. Downloaded on February 03,2021 at 11:32:29 UTC from IEEE Xplore. Restrictions apply.